Transport Layer (In)Security in the Wild

Bernd Kaiser, University of Erlangen-Nuremberg, Germany

***Abstract*—This paper presents the most common possible Transport Layer Security (TLS) weaknesses and attacks, why they exist, and which measures can be taken to complicate or avert their exploitation. The attacks include Beast, Lucky13 and CRIME. We then analyze the Alexa 10,000 most visited Internet websites for vulnerable configurations and show that a high amount of sites (>50%) are vulnerable to one or more of the described attacks. Followed by a short summary of options how to secure TLS connections. We conclude with a examination of TLS' trust model. The focus of this work is on TLS in combination with HTTP, but most parts are relevant to other application areas too.**

***Index Terms*—attacks against TLS, TLS configuration errors, TLS security, TLS configuration statistics, model of trust, certificate authorities**

I. Introduction and Roadmap

Nowadays the Transport Layer Security (TLS) is used everywhere. Most applications use it for secure communications on the Internet, but often those communication are only seemingly secure. TLS is a complex protocol, because most of its components are cryptographic components, and the various implementations are very difficult to use. This background yields many different points of failure. Web browsers are the reason why TLS' ancestors were created in the first place and they are still one of its most important field of application. Hence, in this work the focus will be on browsers and their underling protocol, the Hypertext Transfer Protocol (HTTP).
Section II will show the basic functionality of TLS and Sections III - VI how applications usually make their communications vulnerable to exploitation. Those exploits are explained in detail and for each one a quick fix to improve security is presented. In section VII we also conducted a measurement with the purpose to find out how many websites are affected. The 10,000 most visited websites and their default TLS configurations are studied in terms of security. Finally, in section IX TLS' current trust model will be looked at from a skeptical point of view.

II. Transport Layer Security basics

TLS is a protocol for secure communication over the Internet. The protocol provides a public key infrastructure for authentication and a symmetric encryption for client/server applications to counter eavesdropping, tampering, or message forgery [1]. TLS is an Internet Engineering Task Force (IETF) standard protocol, which was firstly released in 1999 as an upgrade to the Secure Sockets Layer (SSL) V3 protocol [2]. SSL was developed by the Netscape Communications Corporation, but has never been published. In 1995 SSL Version 2.0 was published [3], which contained many security shortages, but was quickly replaced by Version 3.0 in 1996 [4]. In 2008 the current TLS version 1.2 was specified by the IETF in RFC5246 [1].

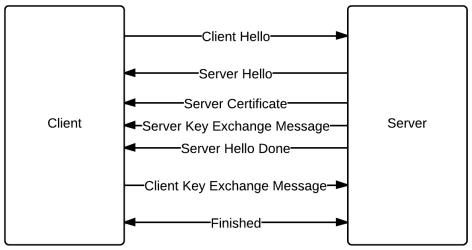

Fig. 1. TLS handshake

TLS is widely used in the World Wide Web (WWW). Every major web browser and most chat, e-mail, and voice-over-ip applications support TLS.

To start a TLS communication, a so called TLS handshake is performed. If the handshake was successful, the client and server applications can be certain that all TLS criteria are fulfilled. If there is an error during this procedure, the handshake fails and no connection is established.

The handshake without client authentication and TLS extensions according to RFC5246 [1, pp. 37-63] (as shown in Figure 1):

1) Client Hello:
   This is the first message sent from the client to the server. It contains the protocol version, the current time and date as a 32-bit unix timestamp (seconds elapsed since 1970-01-01T00:00:00Z), 28 random bytes provided by a secure random number generator, an optional session identifier, a list of supported compression algorithms and a cipher suite list. The cipher suite list contains all the client's supported cipher suites. Each cipher suite contains a key exchange algorithm (e.g., RSA, Diffie-Hellman), a pseudorandom function family, a message authentication code (MAC) algorithm (e.g., SHA1, MD5), and a bulk encryption algorithm (e.g., RC4, AES). The first entry in the cipher suites list is the client's preferred cipher combination and the last entry the most undesired combination.
2) Server Hello:
   Upon receiving the client's hello message, the server checks the cipher suite list and determines if there are any cipher suites it supports as well. If there are no matching cipher suites, the server will notify the client about a handshake failure. Otherwise a reply message will be send.
3) Server Certificate:
   Directly afterwards the server sends its public key certificate followed by every certificate which is needed

to verify the chain of trust to its root certificate. The root certificate is not provided by the server, but must be provided separably. The validation of this certificate chain has to be done by the client's TLS implementation. It's not part of the handshake specification. It is also important to check if the actual hostname and the hostname supplied by the certificate are the same.

4) Server Key Exchange Message:
   This message is sent only if additional data for the premaster secret is needed. For example if a Diffie-Hellman key exchange algorithm is selected, the server sends its public value signed with its private key.
   It is not allowed to send a Server Key Exchange Message for RSA.
5) Server Hello Done:
   The server is done with the key exchange.
6) Client Key Exchange Message:
   The client sets the premaster secret for the TLS connection. If RSA is used, said secret is encrypted with the server's public key. If Diffie-Hellman is used, the client sends its public value.
7) Finished:
   The new shared secret is used for the connection. Both parties send a `Change Cipher Spec` message, before sending the Finished message containing a hash of all handshake messages. Both sides check if they received the same hash. This protects against a Man-in-the-middle attack, which tries to alter the cipher suites or protocol versions. The handshake is complete and all following messages must be encrypted with the shared secret. Server and client have to verify this condition.

One important part of this procedure is how to decide which public keys to trust and which not. In gernal, this is done by trusting certain so called Certificate Authorities (CA) [5], which are normally private companies or government agencies, which sign public certificates for mostly paying costumers only after verifying their identify. Sometimes those CAs also delegate their signing capability to sub-CAs.

Which CA to trust is totally up to the application. Sometimes applications decide to trust the operating system's CAs (e.g., Google Chrome), or supply their own list (e.g., Mozilla Firefox) [6], [7].

III. Authentication

Authentication plays a significant role in TLS. The best connection encryption is useless if the communication partner is not the expected one. TLS uses a public key infrastructure (PKI) for validation.

A. Certificate validation

In general, certificate validation should be done after the `Server Hello Done` message. Every certificate in the signing chain, up to the known root certificate, has to be checked if it is valid and allowed to sign certificates in its parents name.

In 2012 Martin Georgiev et al. published the paper "*The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software*" [8], in which they looked at certificate validation and host validation outside of web browsers. Their attack consisted of a Man-in-the-middle with self-signed, third-party, and valid (signed by a CA) certificates. This attack would be in vain if the developers used TLS correctly on their servers [8, p. 2]. Their verdict was that TLS validation is done wrong in all kinds of application (payment APIs like Amazon Flexible Payments Service, PayPal Payments Standard and PayPal Invoicing, instant messengers, a mobile banking app). They identified that the reasons for this problem are the complicated TLS implementation APIs. Most applications do not use TLS implementations like OpenSSL or GNUTLS, but programming language functions or other frameworks. So even if the developers try to enable validation, they use a wrong function parameter and disable it by accident. For example, a vulnerable PHP implementation of Amazon Flexible Payments Service used cURL's `CURLOPT_SSL_VERIFYHOST` with the boolean value *true* for the parameter, which is automatically cast to *1*, but has to be *2* to enable `CURLOPT_SSL_VERIFYHOST` [8, p. 5].

Other notable examples are the widely used PHP function `fsockopen` and the standard python packages `urllib`, `urllib2`, and `httplib` [8, p. 5].

At least Python clearly states this in a red warning box in its user manuals and provides a standard package `ssl` that verifies certificates. Hostname checking must be done manually. Nevertheless most Python applications do not use this package [8, pp. 9-10].

But not all developers try to get certificate authentication working. Martin Georgiev et al showed that many developers deliberately disable certificate validation, because enabling it is too complicated. These "bad development practices" even get into "critical software responsible for transmitting financial information and sensitive data" [8, pp. 9-10].

The overall state of certificate validation outside of browsers is very bad. Many applications fail using TLS correctly, with too complicated TLS APIs as the reason. There is no easy solution for this problem. The paper suggests easier, and hence completely redesigned TLS implementations, or completely new network communication protocols [8, p. 11]. As long as the old implementations are still the norm, programming languages should assist detecting wrong TLS usage and formal tests should be designed.

B. Certificate revocation

Like most PKI certificates, TLS certificates can be revoked once the owner thinks the private key got compromised. The issuing CA revokes the certificate and adds it to its certificate revocation list (CRL) [9]. The problem with those lists is their distribution. At first, the complete list was published by the CA every day or in similar periodic intervals. The Online Certificate Status Protocol (OCSP) was developed to create a possibility for real time checking of certificate status [10]. OCSP requests are signed and should therefore not be modifiable by an attacker. This service has to be set up by the CA as a central server and is therefore a privacy concern [11, pp. 1-2]. If a OCSP request is sent, the server can learn which websites the client wants to visit.

Another even more serious problem is that a Man-in-the-middle attacker could easily filter the requests to the central OCSP/CRL services [12]. In this case the client is no longer able to check for revocations and might continue with the handshake. If clients would just stop the handshake, a denial of service attack against the CA's OCSP/CRL service would end all TLS handshakes.

A new approach for certificate revocation was introduced by Adam Langley, a Google researcher for the Google Chrome web browser, in 2012 [13]. Langley suggested that the CAs push their CRLs to a list crawlable by Google. Only certificate revocations for security reasons, not administrative reasons, are included in this list. Google will henceforth include this revocations in its regular updates. An attacker could still block Chrome updates, but he has to block them all the time. From the day of the revocation to the day of the attack.

In this scenario the application developers have to release frequent updates, but the attacks get way more difficult.

IV. Attacks against cryptography

> *"The system is only as strong as the weakest key exchange and authentication algorithm supported, and only trustworthy cryptographic functions should be used."*
> —RFC 5246, TLS Version 1.2 [1, p. 96].

As seen in the previous sections, TLS consists of different cryptography components. If only one fails, the integrity of TLS is lost. The next subsections show how this can happen and highlights the cryptography practice, Perfect Forward Secrecy, which gives additional security for recorded messages after private keys used for authentication have been compromised.

A. Weak cipher suites

The following ciphers respectively cipher suites and key sizes should not be used anymore, because of known attacks [14]:

- All SSLv2 cipher suites.
- Export (EXP) level cipher suites in SSLv3.
- RSA or DSA keys smaller than 1024 bits.
  Smaller keys are factorable (see section IV-E2).
- Symmetric encryption algorithms with smaller keys than 128 bits.

If one of these suites, or an old protocol version is needed for backward compatibility, client applications should provide an option for enabling them only for specific resources. Google Chrome has this kind of whitelisting for SSL Version 2 [15].

B. BEAST Attack

The BEAST attack was introduced by Juliano Rizzo and Thai Duong in 2011 [16]. It targets TLS implementations with Version 1.0, which use bulk ciphers with cipher block chaining (CBC), for example the Advanced Encryption Standard (AES), and is a client-side attack [17]. CBC ciphers `xor` every plaintext block with the output of the previously encrypted block before the block is encrypted [18]. The first plaintext block is `xor`ed with an additional random data block, the initialization vector (IV).

The TLS protocol before version 1.1 made the mistake to reuse the last sent block as starting IV for the next TLS message. Rizzo and Duong used this vulnerability to steal web browser session cookies sent through a TLS secured connection. The following example illustrates an attack: a Java applet gets injected into the browser by visiting an evil website, or through a Man-in-the-middle attack on any unsecured web page request. To steal a session cookie, the applet has to know the cookie's name and the length of its value. This is no problem, since many websites use hashes with a fixed length for the cookie value. The applet then tries to split the request in two blocks. The first block only contains one byte of the cookie, the rest is filled with user-controlled data. Now it tries to recreate this output again by guessing this one byte. If it is guessed correctly, the next block will look like the last block. Then the applet moves its boundary, includes the recreated byte and searches for the second byte of the cookie. This is done until the whole cookie is recovered.

This mistake was addressed in TLS 1.1, where every TLS message creates a new random IV [19]. But many servers still support TLS 1.0 for compatibility with older clients. Most TLS implementations also fixed this vulnerability for TLS 1.0, but not all vendors.

Before this attack was published, few applications used TLS 1.1 [20]. As of 2013, the major web browsers support TLS 1.1 [21].

C. Lucky 13

In February 2013 Nadhem AlFardan and Kenneth Paterson released another working attack against CBC-mode encryption and all current TLS versions were affected. The attack was published as "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols" [22].

Lucky Thirteen is a timing side channel attack. It is possible because of different er answer times for messages with bad padding. The possibility for this attack was actually well-known during the specification of TLS v1.1 and v1.2 and mentioned in the RFC, but the small timing difference was conspired not exploitable [1, p. 24]. The authors were able to measure these differences and, after a statistical analysis of roughly 2^{23} multi-session samples, regain the plaintext [22, p. 2]. If their attack with the techniques introduced by the BEAST attack for the cookie decryption, they were able to reduce the sample size to 2^{13} [22, pp. 8-9].

Most major TLS implementations were affected [22, pp. 9-15]. AlFardan and Paterson disclosed the vulnerability to manufacturers before publishing their paper. Nearly all involved parties had already released a fix for this attack, when the paper was publicly available [22, p. 3]. This shows it is very important to always keep the TLS implementations up to date and move away from CBC ciphers.

D. On the security of RC4 in TLS

After the publishing of the BEAST attack and before Lucky13 most sites switched to the streaming cipher RC4

[23]. For nearly two years RC4 was the recommended bulk cipher, because of its wide availability in all TLS Versions. But their were already known weaknesses of RC4 for quite some time [24]. Bernstein et al. showed they could discover cookies in cipher texts using the known RC4 biases [25]. They have not released their precise technique, yet .

Like CBC ciphers, RC4 should no longer be used.

E. Factoring private keys

Nadia Heninger et al. showed in 2012 that even if the cryptography ciphers are solidly working in theory, the private keys can still be recoverable. They published the paper "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices" [26]. They tried to recreate RSA and DSA private keys by factoring different public keys. The rest of this section will focus on RSA keys, because they are most frequently used in TLS.

1) RSA basics: An RSA public key is a pair of integers: n, the modulus, and an integer e, the exponent. n is calculated from the multiplication of two prime numbers p and q [27]. To encrypt a message m into a cipher text c the following computation is done:

$$c = m^e \bmod n$$

For the decryption, p and q are needed. Computing the decryption exponent d:

$$d = e^{-1} \bmod (p-1)(q-1)$$

To decrypt the cipher text c back into the message m:

$$m = c^d \bmod n$$

So if an attacker could factor the public key n, decryption would be possible. Until now, no one was able to factor a properly created 1024-bit RSA key. A 768-bit RSA key has been factored in 2009 [28]. It took nearly a year of distributed computing. No 1024-bit key has been broken yet, but the National Institute of Standards and Technology (NIS) recommends using keys with 2048 or more bits [29].

2) Factor similar RSA keys: Factoring one public key is not practical, but computing the greatest common divisor (GCD) of two 1024-bit integers is practical [26, p. 2]. So if two public keys n_1 and n_2 have one common prime number $p_1 = p_2 = p$, this prime factor can be calculated in microseconds:

$$p = gcd(n_1, n_2)$$

If the common factor p is calculated, the two other primes q_1, q_2 can be easily computed by division:

$$q_1 = \frac{n_1}{p}$$

$$q_2 = \frac{n_2}{p}$$

Now both private keys are compromised.

3) Mining public keys: To get enough key samples for the following tests, Nadia Heninger et al. mined the complete IPv4 space, certain special address ranges excluded, for public keys [26, pp. 3-4]. In this process they acquired 12,828,613 TLS keys. It took ~25 hours to discover all TLS servers and ~96 hours to fetch the public keys.

4) Repeated keys: In this data many public keys were exactly the same and not owned by the same company or webhoster. One reason were default keys (5.23%) of headless devices, for example routers or embedded hardware [26, pp. 6-7]. After the assembly of those devices, a standard firmware is installed, which contains the same private key for all devices. Those private keys could be obtained by reverse engineering the firmware. There are already published databases containing device names and default private keys.

Another reason for repeated keys (0.34%) is that many devices create their private key with low entropy [26, pp. 7-8]. This mostly happens when they create the keys right after the boot process [26, p. 12]. The entropy in their pseudo random number generators (PRNG) is not high enough and therefore they often produce the identical key pairs.

5) Factorable keys due to low entropy: 23,576 (0.40%) TLS public keys were found, whose private key could be factored [26, pp. 8-9]. Two of those keys have been signed by a CA. Most keys (99%) could be assigned to headless devices manufactured by 41 companies.

Once again low entropy was the problem, but unlike with repeated keys, only one prime factor was recreated [26, pp. 8-9]. An identical p is computed with the shared random value. After this computation new random values are generated, for example by a clock tick. Now a distinct q is calculated, because of the higher entropy.

6) Counter measures: For (headless) device manufactures the following defenses and lessons were presented [26, pp. 17-18]:

- No default keys or certificates.
- Seed PRNG with truly random data at assembly.
- Ensure the existence of effective entropy sources for the PRNGs.
- Test randomness upon completion of the device.

Operating system developers were assigned the task to provide PRNGs with enough entropy for the TLS libraries on any platform (desktop computer or headless router) [26, p. 17]. The TLS library developers should use these secure PRNGs and not inadequate ones. For example, many Linux security applications, like OpenSSL, use `/dev/urandom` and not `/dev/random` [26, pp. 15-16], despite the fact that even the Linux user manual explicitly warns against this usage for security applications [30].

They also developed an online service[1] where system administrators can check if their public keys are affected [26, p. 8].

[1] https://factorable.net/keycheck.html

F. Perfect forward secrecy

Most TLS servers use a cipher suite, which uses the servers public certificate for encryption of the premaster secret for the symmetric encryption. As shown in the previous paragraphs these private keys can get compromised or can be attackable from the start. Another important philosophy in cryptography is that you do not expect your ciphers to be secure for ever. In 10 years, somebody could come up with a new attack and break it.

So if you used the same private/public key pair for authentication and for the transmission of the symmetric key, an attacker with the private key could not only fake authentication, but also decrypt every communication recorded in the past [31, p. 1].

Perfect forward secrecy (PFS) prevents this easy decryption [31, p. 7]. PFS is achieved with the Diffie-Hellman key exchange.

In 1977 Martin E. Hellman et al. filed for a patent for an exchange procedure of cryptographic keys [32]. The so called Diffie-Hellman key exchange (D-H) allows the creation of a shared secret over an insecure communication channel. This secret will be used as the symmetric key. The patent is now expired and the technique is free to use.

D-H uses the discrete logarithm problem and power associativity to achieve this. A D-H key exchange between Alice and Bob is performed as follows [33]:

1) Alice and Bob agree on a prime number p and a base g. These values are public.
2) Alice and Bob pick a random integer a and b. Only Alice knows a and only Bob knows b.
3) Alice computes
$$A = g^a \bmod p$$
and sends A to Bob. A is now publicly known.
4) Bob computes
$$B = g^b \bmod p$$
and sends B to Alice. B is now publicly known.
5) Alice computes
$$s = B^a \bmod p$$
and knows now the secret s.
6) Bob computes
$$s = A^b \bmod p$$
and knows now the same secret s.

The secret s is the same, because $(g^a)^b$ and $(g^b)^a$ are equal (see power associativity). a and b are discarded after every session. No two communications use the same shared secret.

In TLS these D-H values are send in the `Server key exchange` message and the `Client key exchange` message [1, pp. 50-53, 61]. The server also signs it value with its private key of the authentication certificate. The client verifies the value with the previously received and checked certificate.

Many TLS implementations, like OpenSSL, support D-H for the premaster secret exchange [34]. A relative performant cipher is `ECDHE-RSA-RC4-SHA` (D-H with elliptic curves) [35]. RC4 as bulk cipher is used, because TLS Versions smaller than 1.1 with bulk cipher block chaining ciphers are vulnerable to the BEAST attack (see section IV-B). The performance is only good compared to other D-H key exchange, RSA-only is much faster [36]. This may be one reason why few TLS servers use PFS. Nevertheless Google started using PFS for most of its services back in 2011 [37].

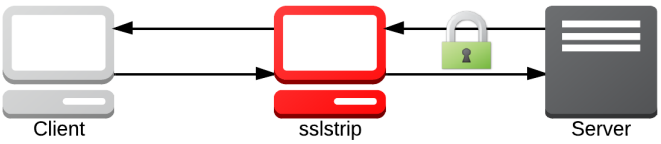

Fig. 2. sslstrip Man-in-the-middle attack

V. TLS Stripping

Most web browser users do not explicitly connect to HTTPS servers. Normally they just request a domain like `www.example.com`. The browser implicitly adds the HTTP protocol and connects to `http://www.example.com`. Then the web server redirects the client with a HTTP status code (302 Found Location) to `https://www.example.com`.

At Black Hat 2009 Moxie Marlinspike showed this behavior could be used to downgrade HTTPS to HTTP connections [38]. He also released a working Man-in-the-middle attack tool: `sslstrip` [39]. `sslstrip` uses `arpspoof` to redirect the targets (has to be in the same (wirless) local area network) network packages to the attacker [40]. After `sslstrip` hijacked the communication, it monitors every http request. If a server sends a hyperlink <a>, or a HTTP header location with a HTTP**S** address, `sslstrip` will turn it into a HTTP request [38, p. 57]. Every time a stripped down URL is requested, `sslstrip` will proxy this request to the server [38, pp. 58-59]. The connection between `sslstrip` and the server is secured with TLS, as shown in figure 2. This is possible, because in general there is no client authentication, only server authentication. So on the server side everything looks right and on the client side only the HTTPS indication is missing. With another feature of `sslstrip` Marlinspike tries to trick the user in believing their connection is TLS secured. The website's favicon gets replaced by a closed lock [38, pp. 65-67]. In older browser versions a closed lock indicates a secured connection.

To address this vulnerability *HTTP Strict Transport Security (HSTS)* was introduced by the IETF [41]. HSTS is an opt-in security measure for servers. A client activates HSTS on receiving the HTTP response header `Strict-Transport-Security` over a TLS secured connection [41, p. 13]. The response header has the following syntax:

```
Strict-Transport-Security:
max-age=94670000; includeSubDomains
```

max-age is the duration, in seconds (3 years in this case), of how long the client will keep this website in its HSTS list.

If the optional parameter *includeSubDomains* is supplied, all sub domains are included too. As long as a website is in the client's HSTS list, the following security controls will always be enforced:

- All requests will be sent through HTTPS.
- The browser automatically upgrades any HTTP requests to HTTPS.
- TLS certificates have to be valid. If a certificate is not valid, the client's operator cannot override the error.

So if the user visited the website before, the client is not at risk of TLS stripping as long *max-age* seconds have not expired since the last visit. Google Chrome also has a preloaded HSTS list [42]. All sites on this list will always use HSTS. Administrators can request that their websites are added to this list.

Sven Schleier and Thomas Schreiber examined the occurrence of HSTS [43]. They looked at the response headers of all the websites in the "Alexa Top 1,000,000 list", which support TLS [44]. Only 0.4% provide the HSTS response header [43, p. 6]. But most of these few severs deliver a *max-age* duration, which is too short. Just 0.25% HSTS directives guarantee protection for a year or more.

VI. CRIME attack on compressed TLS

As shown in section II, TLS is capable of using compression algorithms for connections. In 2012 Juliano Rizzo and Thai Duong created the CRIME exploit targeting web browsers [45]. It uses plain text injection and inadvertent information leakage caused by compression to steal HTTP-Cookies and hijack sessions.

A. CRIME

For the exploit to work, two conditions, besides TLS compression, have to be met [45, pp. 3-4]:

1) The attacker must be able to sniff the network traffic. For example, by being connected to the same wireless local area network with symmetric encryption (WPA2).
2) Send TLS encrypted requests to the targeted website. This can be achieved by visiting an malicious website, or through a Man-in-the-middle attack by injecting Javascript on an unencrypted website.

If this requirements are met, CRIME uses the length of the compressed TLS packages as oracle. This method was first introduced by John Kelsey in 2002 [46]. Every HTTPS request contains the user input, which is controlled by the attacker, and the session cookie. The attacker has no way in accessing the cookie. In the web browser the cookie is secured by the Same-Origin-Policy, so it is not accessible through Cross Site Scripting, and the HTTPS request itself is encrypted.

Compression makes it possible to store the same amount of data in fewer bits. Deflate's LZ77 compression for example removes redundancy between repeated strings [47]. The repeated strings are replaced with back-references to the last occurrence.

This pseudo code shows how the length of a compressed HTTPS request is calculated [45, p. 13]:

Total websites contacted	10,000	100%
No TLS available	5,106	41.6%
TLS handshake incomplete	755	7.55%
SSL Version 3	39	0.39%
TLS Version 1.0	3,572	35.27%
TLS Version 1.1	1,528	15.28%
TLS Version 1.2	1799	17.99%

TABLE I
TLS AVAILABILITY WITH SPECIFIC PROTOCOL VERSIONS IN THE ALEXA TOP 10,000 WEBSITES LIST.

Total successful TLS connections	5,130	100%
Valid certificates	4,265	83.0%
Strong public keys ($\geq$2048-bit)	3,413	66.41%
Perfect forward secrecy	1,651	32.13%
Weak cipher suites	512	0.01%
Weak public keys ($<$1028-bit)	11	0.002%
Vulnerable against CRIME (Compression enabled)	1,390	27.05%
Vulnerable against BEAST (CBC and $<$TLSv1.1)	2,427	47.23%

TABLE II
PROPERTIES OF SUCCESSFUL TLS CONNECTIONS

$$httpRequestLength = length(compress(input + cookie))$$

So if there is more redundancy between the user input and the cookie, the request will be smaller as a request with totally different user input and cookie value.

This characteristic is used to guess the secret cookie by submit different user input and sniff for the smallest HTTPS request [45, pp. 19-22]. So the secret cookie can be guessed piece for piece with the HTTPS request's length as oracle.

B. Counteractive measures

How to repel this attack is obvious: disable TLS compression [48, p. 4]. The user input itself can still be compressed by the server and the client. Only the HTTPS headers must not be compressed.

HTTPS header compression has been removed by all affected browser vendors after this attack was disclosed to them [49]. Older browser versions are still vulnerable to CRIME. But not only the client can stop requesting compression in its `Client Hello` message, the web server is also able to request a non compressed connection. So server developers like the Apache Foundation disabled TLS compression by default [50]. No server should still use or support it.

VII. TLS configuration comparison of Alexa top 10,000 websites

After presenting all these different attack scopes, we tried to identify how many websites are still vulnerable. The Alexa "Top 1 Million list" served as website source for this measurement. The first 1000 sites were selected and then a TLS connection was attempted for each one with `OpenSSL` in version 1.0.1c (current stable release) [51], but only with one cipher

suite, which the server chose. So this measurement reflects the preferences of the server. Ubuntu's, in Version 12.10, selection of root-CAs was trusted for public key authentication.

Table I and table II show the results.

Around 51% web server supported TLS, some other sites may use TLS after all. For example eBay does not support TLS on `https://ebay.com`, but on `https://signin.ebay.com`. These sites are vulnerable to TLS stripping (see section V).

The sites who directly allowed TLS connections mostly used TLSv1.0 and a good portion of about 15% TLSv1.1. Just 39 served only SSLv3 and not SSLv2 at all. But almost half of all TLSv1 servers used a CBC cipher so that client's could be exploited with the over one and a half year old BEAST attack. Its successor CRIME is also widely usable. 27% still use compression. Weak cipher suites are almost no problem and factorable public keys are virtually nonexistent. Over four fifths of all public keys are properly signed and could be verified. The rest were self-signed or are already expired. But even if one authentication certificate would be compromised, the past communications of nearly a third of the servers would is save, because they use perfect forward secrecy.

All in all, these measurements show that TLS security awareness needs to be strengthened. Mostly, trivial changes like disabling compression or adding a response header already boost security considerably.

VIII. Short summary about how to increase TLS security

	Section
Verify certificates	III-A
Do not use weak cipher suites or insufficient public key sizes	IV-A
Do not use CBC cipher suites	IV-B & IV-C
Do not use RC4 cipher suites	IV-D
Check for factorable keys	IV-E2
Only serve TLS connections; Enforce with HSTS	V
Disable Compression	VI

TABLE III
Configuration options to increase (or create) TLS security

Table III is a brief summary about how to increase TLS security and which section of this document explains why these actions are required.

IX. Trust issues

> *"Implementations and users must be careful when deciding which certificates and certificate authorities are acceptable; a dishonest certificate authority can do tremendous damage."*
> —RFC 5246, TLS Version 1.2 [1, p. 96].

The central issue of the TLS trust model is how to *verifying the identity of the public key owner*. As mentioned before, this is done be trusting certain certificate authorities (CA). For example, Microsoft Windows fully trusts 266 different root CAs and Mozilla Firefox allows 144 CAs to sign any certificate for any domain [52, p. 3]. As long as every CA and their sub-CAs do an honest job everything is fine. But only one dishonest or compromised (sub-)CA can completely break the whole chain of trust.

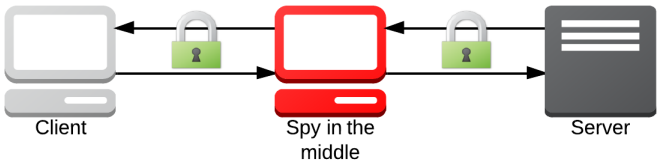

Fig. 3. A Man-in-the-middle attacker with public key signing ability.

Let us jump right to the worst case scenario: A Man-in-the-middle (e.g., intelligence agency) has got a trusted certificate with signing capabilities (Certificate Basic Constraints: Is a Certification Authority). In this case, the attacker is able to spoof every identity for any requested domain or server. For example, the target's client application requests a TLS connection to `example.com`. Thereupon, the attacker creates himself a public key and signs it with his sub-CA key. Now he is able to act as proxy between the target and `example.com`. Both connections are TLS secured as shown in Figure 3. This scenario is not unlikely, because some trusted root CAs are government organizations [52, p. 5].

Christopher Soghoian and Sid Stammy analyzed those attacks in their paper "Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL", and presented strategies to defend against them [52]. The result is `Certlock`, an extension for the Mozilla Firefox web browser. `Certlock` keeps track of the certificates used by a web server. If the browser receives not the identical certificate at the next visit, `Certlock` will analyze the new certificate [52, p. 9]. If the new certificate was issued by the same CA or a CA from the same country, the user will not be noticed about the change. But if the CA's country differs, the user will be warned about the new certificate and is shown where the signing CA is located. This behavior will not protect against attacks from CAs located in the same country [52, p. 10].

Finally, a look at totally different trust model: a Web of Trust [53]. In this trust model, everybody is able to sign public keys. The level of trust for a public key is determined by how many other users signed this key with their private key. Who signed the key could also be taken into the trust consideration. This is a decentralized approach, where companies like CAs or government agencies would have no special powers. It is highly unlikely that they would support such a model shift. Quite the reverse, opposition would be certain.

X. Conclusion

Configure TLS applications correctly is not easy. The different sections of this worked showed that on each component of the protocol configuration errors can be made. But with mostly simple adjustments, many weaknesses can be removed. One easy step is to disable all known weak ciphers, make sure the client actually verifies the authentication, and always

serve content only over HTTPS. This can be enforced with the HSTS HTTP header. Preventing CRIME is also no problem, just disabling TLS compression completely nullifies this attack vector. Also the pre master secret should always be negotiated with perfect forward secrecy. But the questions which bulk ciphers to support is not that easy. The most common types of available ciphers, CBC and RC4, are both vulnerable. But as long as TLS v1.2 is not an universal option, all major web browser do not support v1.2, there is a need to support at least one of them. In this case the TLS implementations must be up to date. If TLS v1.2 is available, a dedicated authenticated encryption algorithm like AES-GCM should be used.

Developing TLS applications is hard. Most TLS implementations are very complicated to use and developers make many wrong assumptions about them. Matthew Green made the following suitable analogy: *"OpenSSL is the space shuttle of crypto libraries. It will get you to space, provided you have a team of people to push the ten thousand buttons required to do so"* [54]. Only easier to use TLS implementation will solve this problem and therefore their development should have the highest priority.

Solving TLS' trust problem is impossible. As long as so many companies and institutions have the power to create certificates which are globally valid, trust breaches are going to happen. The recently fake, but valid, `google.com` certificate underlines this statement [55].

Overall, our transport layers are far from being secure.

XI. Acknowledgments

Thanks to Sebastian Schinzel, Fabian Meyer, and Anton Kaiser for their input and feedback.

References

[1] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, updated by RFCs 5746, 5878, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[2] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246 (Proposed Standard), Internet Engineering Task Force, Jan. 1999, obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt

[3] Wikipedia, "Transport layer security — wikipedia, the free encyclopedia," 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=531977019

[4] A. Freier, P. Karlton, and P. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101 (Historic), Internet Engineering Task Force, Aug. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6101.txt

[5] Wikipedia, "Certificate authority — wikipedia, the free encyclopedia," 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Certificate_authority&oldid=525349728

[6] "Google chrome root certificate policy." [Online]. Available: http://dev.chromium.org/Home/chromium-security/root-ca-policy

[7] Mozilla Foundation, "Included certificate list," Jun. 2009. [Online]. Available: http://www.mozilla.org/projects/security/certs/included/

[8] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.

[9] R. Housley, W. Polk, W. Ford, and D. Solo, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 3280 (Proposed Standard), Internet Engineering Task Force, Apr. 2002, obsoleted by RFC 5280, updated by RFCs 4325, 4630. [Online]. Available: http://www.ietf.org/rfc/rfc3280.txt

[10] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP," RFC 2560 (Proposed Standard), Internet Engineering Task Force, Jun. 1999, updated by RFC 6277. [Online]. Available: http://www.ietf.org/rfc/rfc2560.txt

[11] M. Narasimha, J. Solis, and G. Tsudik, "Privacy-preserving revocation checking," *International Journal of Information Security*, vol. 8, no. 1, pp. 61–75, 2009.

[12] cemp, "Ocsp: this fail brought to you by the number three," Jul. 2009. [Online]. Available: http://randomoracle.wordpress.com/2009/07/31/ocsp-this-fail-brought-to-you-by-the-number-three/

[13] A. Langley, "Revocation checking and chrome's crl," Feb. 2012.

[14] T. O. W. A. S. Project, "Testing for ssl-tls (owasp-cm-001)," Jan. 2013. [Online]. Available: https://www.owasp.org/index.php/Testing_for_SSL-TLS_(OWASP-CM-001)

[15] "Google chrome ssl settings." [Online]. Available: http://googlechrometutorial.com/google-chrome-advanced-settings/Google-chrome-ssl-settings.html

[16] J. Rizzo and T. Duong, "BEAST," Sep. 2011. [Online]. Available: http://vnhacker.blogspot.de/2011/09/beast.html

[17] N. C.-A. System, "National vulnerability database (nvd) national vulnerability database (cve-2011-3389)," Jun. 2011. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-3389

[18] Wikipedia, "Block cipher modes of operation — wikipedia, the free encyclopedia," 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Block_cipher_modes_of_operation&oldid=530908784

[19] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346 (Proposed Standard), Internet Engineering Task Force, Apr. 2006, obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176. [Online]. Available: http://www.ietf.org/rfc/rfc4346.txt

[20] N. Bolyard, "Support new revisions to tls protocol in psm, once nss does," Apr. 2008. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=422232

[21] Wikipedia, "Comparison of TLS implementations — Wikipedia, The Free Encyclopedia," 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Comparison_of_TLS_implementations&oldid=531932278

[22] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the tls and dtls record protocols," 2013.

[23] M. Green, "Attack of the week: Rc4 is kind of broken in tls," Mar. 2013. [Online]. Available: http://blog.cryptographyengineering.com/2013/03/attack-of-week-rc4-is-kind-of-broken-in.html

[24] P. Sepehrdad, S. Vaudenay, and M. Vuagnoux, "Discovery and exploitation of new biases in rc4," in *Selected Areas in Cryptography*, ser. Lecture Notes in Computer Science, A. Biryukov, G. Gong, and D. Stinson, Eds. Springer Berlin Heidelberg, 2011, vol. 6544, pp. 74–91. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19574-7_5

[25] D. Bernstein, N. AlFardan, K. Paterson, B. Poettering, and J. Schuldt, "Attack of the week: Rc4 is kind of broken in tls," Mar. 2013. [Online]. Available: http://www.isg.rhul.ac.uk/tls/index.html

[26] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your Ps and Qs: Detection of widespread weak keys in network devices," in *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[27] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[28] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. Osvik *et al.*, "Factorization of a 768-bit rsa modulus," *Advances in Cryptology–CRYPTO 2010*, pp. 333–350, 2010.

[29] "Nist sp 800-78-3, cryptographic algorithms and key sizes for personal identity verification," 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-78-3/sp800-78-3.pdf

[30] "random, urandom - kernel random number source devices," Website, 08 2010. [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man4/random.4.html

[31] W. Diffie, P. Oorschot, and M. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992.

[32] S. C. Martin E. Hellman, B. C. Bailey W. Diffie, and P. A. C. Ralph C. Merkle, "Cryptographic apparatus and method," Patent US 4 200 770, 04 29, 1980. [Online]. Available: http://www.patentlens.net/patentlens/patent/US_4200770/en/

[33] Wikipedia, "Diffiehellman key exchange — wikipedia, the free encyclopedia," 2013. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Diffie%E2%80%93Hellman_key_exchange&oldid=532355914

[34] "OpenSSL: Documents, SSL_CTX_set_tmp_dh_callback(3)." [Online]. Available: http://www.openssl.org/docs/ssl/SSL_CTX_set_tmp_dh_callback.html#NOTES

[35] V. Bernat, "Ssl/tls and perfect forward secrecy," 2011. [Online]. Available: http://vincent.bernat.im/en/blog/2011-ssl-perfect-forward-secrecy.html

[36] N. Mavrogiannopoulos, "The price to pay for perfect-forward secrecy," Dec. 2011. [Online]. Available: http://nikmav.blogspot.de/2011/12/price-to-pay-for-perfect-forward.html

[37] A. Langley, "Protecting data for the long term with forward secrecy," Nov. 2011. [Online]. Available: http://googleonlinesecurity.blogspot.com.au/2011/11/protecting-data-for-long-term-with.html

[38] M. Marlinspike, "Blackhat dc 09 - defeating-ssl," Feb. 2009. [Online]. Available: http://www.blackhat.com/presentations/bh-dc-09/Marlinspike/BlackHat-DC-09-Marlinspike-Defeating-SSL.pdf

[39] ——, "Software sslstrip," Feb. 2009. [Online]. Available: http://www.thoughtcrime.org/software/sslstrip/

[40] W. Lai, "Arpsoof," 2009. [Online]. Available: http://arpspoof.sourceforge.net

[41] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," RFC 6797 (Proposed Standard), Internet Engineering Task Force, Nov. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6797.txt

[42] "Http strict transport security - the chromium projects." [Online]. Available: http://dev.chromium.org/sts

[43] S. Schleier and T. Schreiber, "Aktuelle verbreitung von http strict transport security," Nov. 2012. [Online]. Available: http://www.securenet.de/fileadmin/papers/HTTP_Strict_Transport_Security_HSTS_Whitepaper.pdf

[44] Alexa Internet, Inc, "The top 1,000,000 sites on the web." [Online]. Available: http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

[45] J. Rizzo and T. Duong, "The crime attack."

[46] J. Kelsey, "Compression and information leakage of plaintext," in *Fast Software Encryption*. Springer, 2002, pp. 95–102.

[47] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951 (Informational), Internet Engineering Task Force, May 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt

[48] K. Shimizu and B. Kihara, "Considerations for protocols with compression over tls," Oct. 2012. [Online]. Available: http://tools.ietf.org/html/draft-kihara-compression-considered-harmful-01

[49] D. Fisher, "Crime attack uses compression ratio of tls requests as side channel to hijack secure sessions," Sep. 2012. [Online]. Available: http://bit.ly/Npi5Ub

[50] The Apache Foundation, "Apache module mod_ssl documentation." [Online]. Available: http://httpd.apache.org/docs/2.4/mod/mod_ssl.html#sslcompression

[51] "Openssl version 1.0.1c." [Online]. Available: http://www.openssl.org/source/openssl-1.0.1c.tar.gz

[52] C. Soghoian and S. Stamm, "Certified lies: Detecting and defeating government interception attacks against ssl (short paper)," *Financial Cryptography and Data Security*, pp. 250–259, 2012.

[53] A. Abdul-Rahman, "The pgp trust model," in *EDI-Forum: the Journal of Electronic Commerce*, vol. 10, no. 3, 1997, pp. 27–31.

[54] M. Green, "The anatomy of a bad idea," Dec. 2012. [Online]. Available: http://blog.cryptographyengineering.com/2012/12/the-anatomy-of-bad-idea.html

[55] A. Langley, "Enhancing digital certificate security," Jan. 2013. [Online]. Available: http://googleonlinesecurity.blogspot.de/2013/01/enhancing-digital-certificate-security.html